

CS1010S Programming Methodology

Lecture 6

Working with Sequences

19 Sep 2018

Midterm Exam

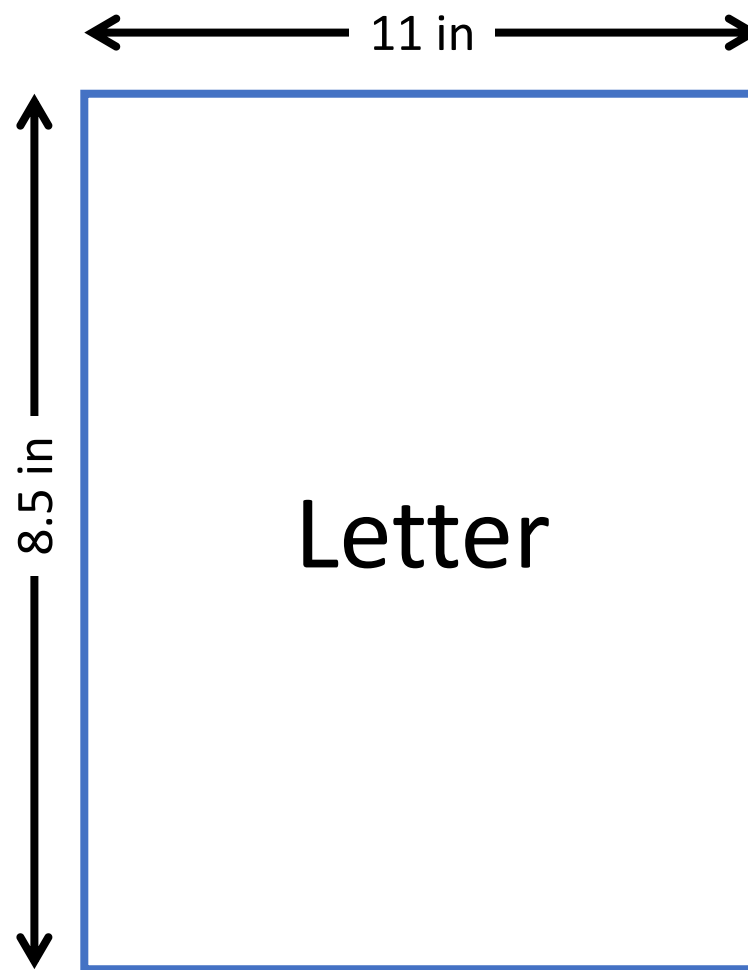
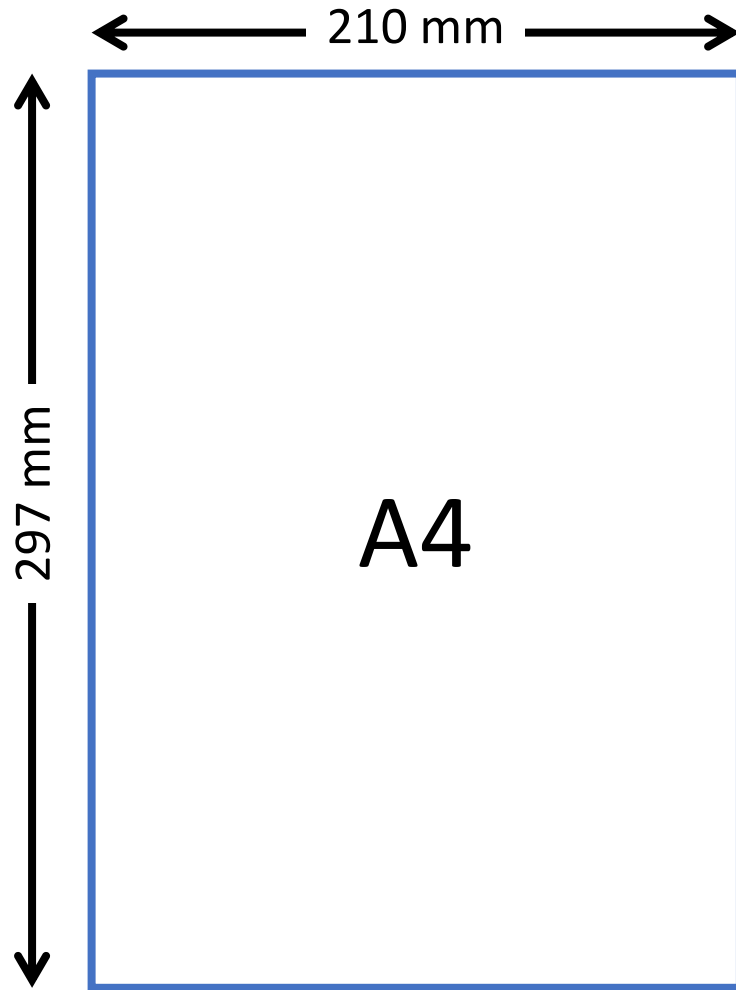
- Date: 3 October
- Time: 6:45pm – 8:15pm
- Venue: MPSH 2



Midterm Exam

- Date: 3 October
- Time: 6:45pm – 8:15pm
- Venue: MPSH 2
- Open-sheet exam (no laptops!)
 - 1 x A4 sheet (both sides)

Cheat Sheet



Midterm Exam

- Date: 3 October
- Time: 6:45pm – 8:15pm
- Venue: MPSH 2
- Open-sheet exam (no laptops!)
 - 1 x A4 sheet (both sides)
 - Printed or Handwritten
 - Monochrome

- ordered sequence
 - list
 - tuple
- non-modifiable values
 - str by
- key containers, n
 - dictionary dict (key/value associations)
 - collection set

```

identifiers  int
9           int
ded         int
           flo
           rou
           boo
           str

```

```

assignment chr
rep
byt
lis
dic
set
sepa
str
"
and str
+= "
/= "
%= sequ
...

```

-3	-2	-1
2	3	4
30	40	50
3	4	
-2	-1	

```
start slice: end slice
lst[::-1]
lst[::-2]
lst[:] + [1]
# up to end.
# we with del lst
```

parent state
state
parent state
state
next state

indentation

@ configure place of an

```

from math import pi,
sin(pi/
cos(2*pi
sqrt(81
log(e**
ceil(12
floor(1
modules math
decimal, fract

```

```
import sin,
)→0.707...
/3)→-0.4999.
→9.0 ✓
)→2.0
5)→13
.5)→12
statistics, random
ons, numpy, etc. (cf. d
```

- ordered sequence
- key container

```

intifiers
variables,
o
int
exp
int

```

*S- and
 .CO-
 /-
 %-
 inner
 contains.

sequence of items at
item at index
30 40
& return item at index
sort of reverse list

```
1st[: :-1]
1st[: :-2]
1st[:] → [1]
```

I

```
from math import *
sin(pi/4)
cos(2*pi/3)
sqrt(81)
log(e**2)
ceil(12.34)
floor(12.34)
modules math
```

```
import sin,
) -0.707...
/3) → -0.4999
-9.0 ✓
) -2.0
5) -13
-5) -12 as f:
statistics, ra
ons numpy. & f
```

normal
proceeding

```

1 public static bool IsPrime(int n)
2 {
3     if (n < 2) return false;
4     for (int i = 2; i <= n / i; i++)
5     {
6         if (n % i == 0) return false;
7     }
8     return true & 1;
9 }

```

error CS0019: Operator '&' cannot be applied to operands of type 'bool' and 'int'

```

10s) format="Retired"
x: toto else:
11bot x format="Active"
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
```

```

=====
=====
e f:
f:
line
=====

```

- Format *fill_char*
- `<> ^`
- integer: `b`
- float: `e` or `f`
- string: `s`
- Conversion flags

sign =
 sign = space
 sign = char, of
 sign = exponential, f
 sign = a (readable

```

" (1: > 10a
→ '
" (x|x) " .
→ 'ℤ \ 'm'

```

```
format(8, "to  
oto'  
rmat(x='I'm')  
-----  
-maxwidth  type  
with 0  
o octal, x or X hexa-  
or G appropriate (de-  
% percent  
representation)
```

\mathbb{R}^n

Midterm Exam

- Scope: everything up to and including Lecture 6 (Today!)
- Past Year Exams have been uploaded to Coursemology

Midterm Exam

1. Python Expressions
2. Solving Problems with Recursion/Iteration
 - Order of Growth
3. Higher Order Functions
4. Data Abstraction
 - Define new Abstract Data Type + Operations

Makeup Midterm Exam

- If you miss the midterm with valid reason
 - MC, Leave of Absence, etc.
- Makeup Midterm
 - Date: Friday 19 October
 - Time: 6:45pm – 8:15pm

Only 15%

Don't Stress

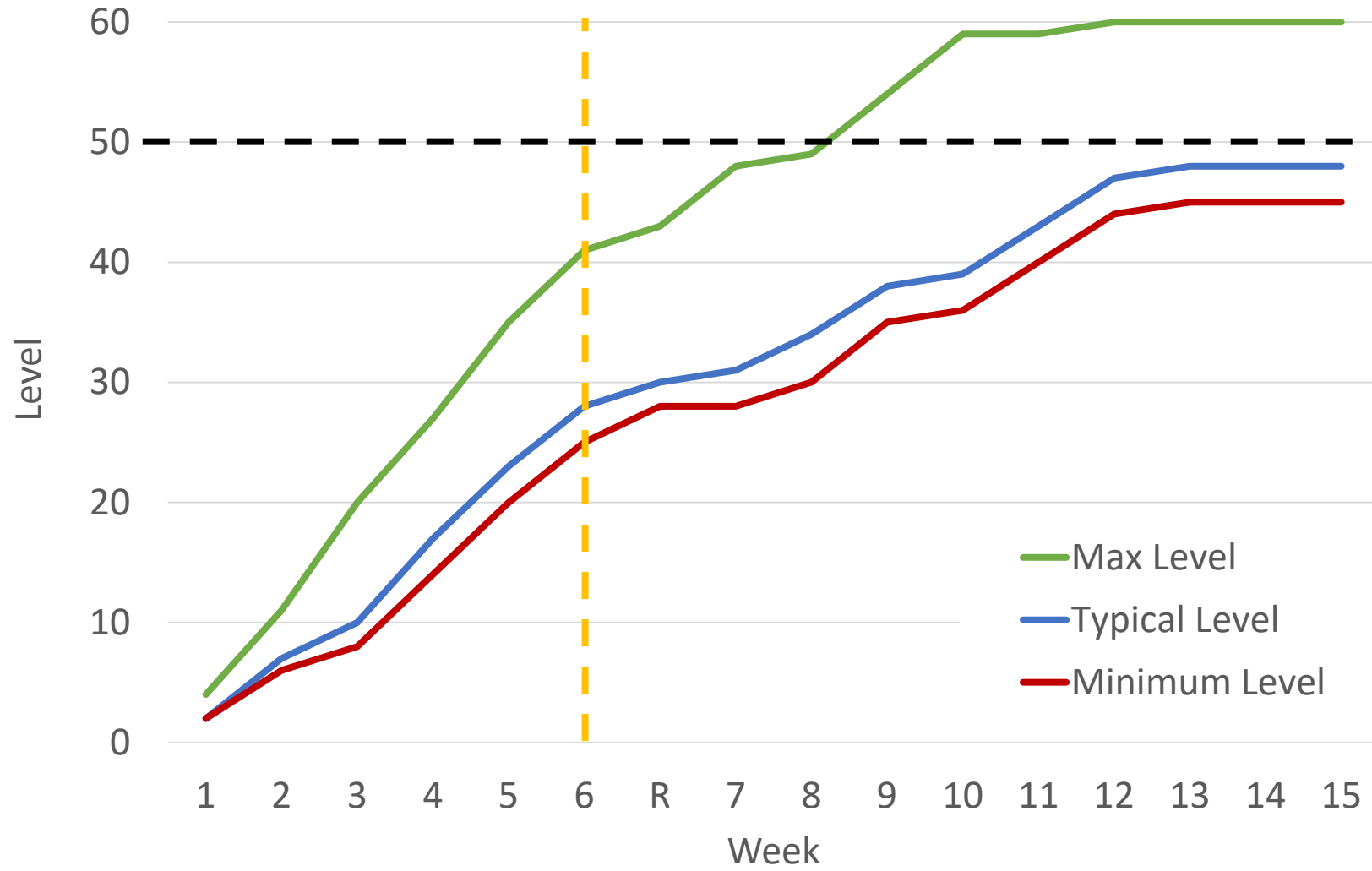
Help is Coming

- Remedial Sessions
 - 19 Sep (Wed), SR8, 6:30 – 8:30 pm
- Past-Exam Review
 - TBA during recess week
- “Desperado” Session
 - 1 Oct (Mon), 6:30 – 8:30 pm
 - 2 Oct (Tues), 6:30 – 8:30 pm

No Tutorials & Recitations

- No Recitations on midterm week
- Tutors will still be in class on Mon/Tues during tutorial times for consultation

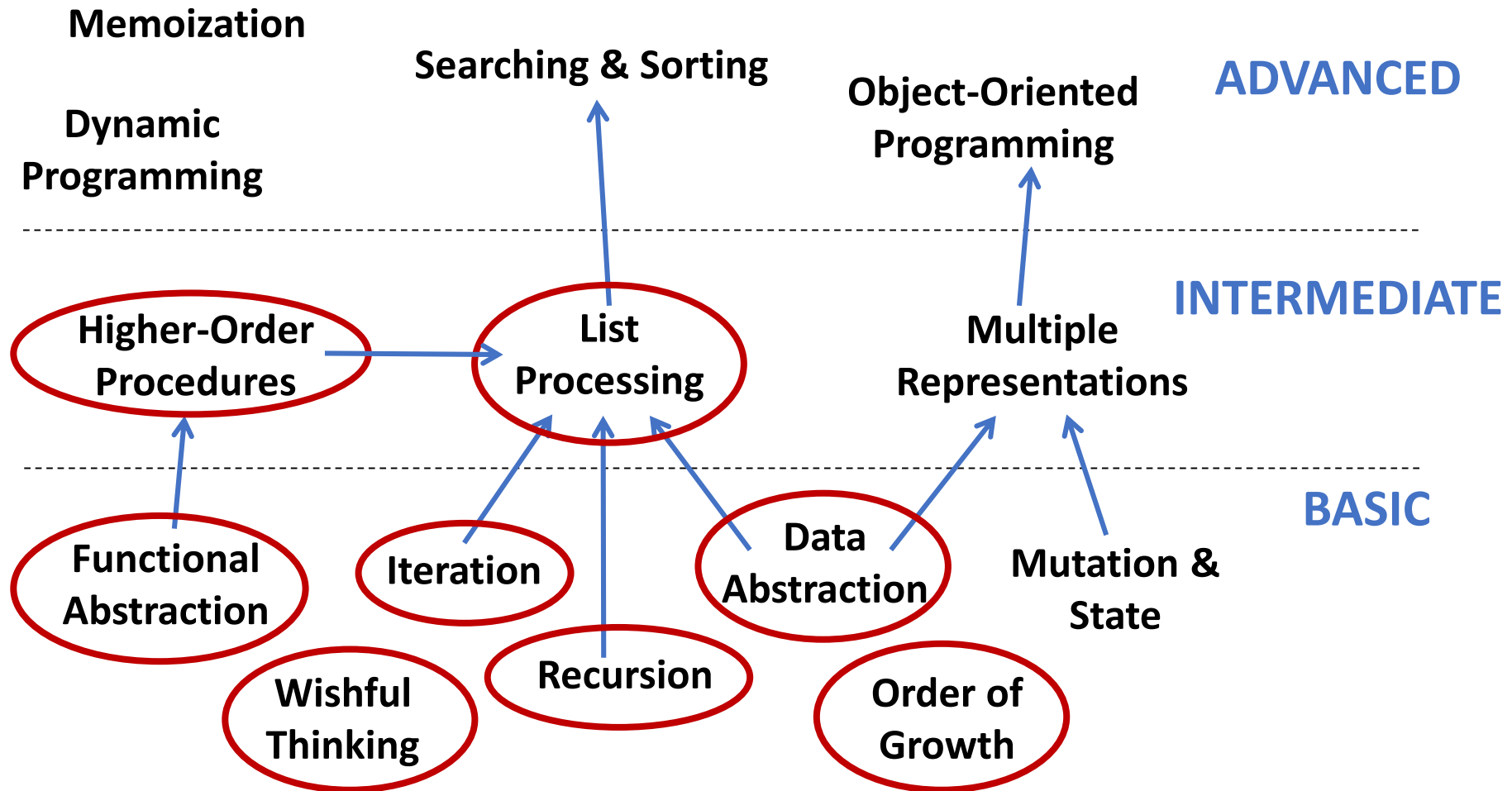
Expected Level Progression



Today's Agenda

- Processing Sequences
 - Recursion & Iteration
- Tree as nested sequences
 - Hierarchical structures
- Signal-processing view of Computations
- Working with Files

CS1010S Road Map



Fundamental concepts of computer programming

Recap: Data Abstraction

- Abstracts away irrelevant details, exposes what is necessary
- Separates usage from implementation.
- Captures common programming patterns
- Serves as a building block for other **compound data**.

Key idea

- Decide on an internal representation of the Abstract Data Type (ADT) **Tuple!**
- Write functions that operate on that new ADT

Key insight: nobody needs to know your internal representation to use your ADT

Guidelines for Creating Compound Data

- Constructors
 - To create compound data from primitive data
- Selector (Accessors)
 - To access individual components of compound data
- Predicates
 - To ask (true/false) questions about compound data
- Printers
 - To display compound data in human-readable form

Sequences

- Sequential data, represented by tuples

- Get the first element of the list:

`seq[0]`

- Get the rest of the elements:

`seq[1:]`

- If a seq. is a tuple containing a single integer 4:

`seq = (4,)`

`seq[0] → 4`

`seq[1:] → ()`

Reversing a Sequence

```
def reverse(seq):  
    if seq == ():  
        return ()  
    else:  
        return reverse(seq[1:]) + (seq[0],)
```

- Notice that `(seq[0],)` is a tuple and not an integer
- Can only concatenate tuples with tuples

Reverse Example

reverse(1, 2, 3, 4)
reverse(2, 3, 4) + (1,)
reverse(3, 4) + (2,) + (1,)
reverse(4) + (3,) + (2,) + (1,)
() + (4,) + (3,) + (2,) + (1,)
(4,) + (3,) + (2,) + (1,)
(4, 3) + (2,) + (1,)
(4, 3, 2) + (1,)
(4, 3, ,2 ,1)

```
def reverse(seq):  
    if seq == ():  
        return ()  
    else:  
        return reverse(seq[1:]) + \  
            (seq[0],)
```

Recursive

Order of Growth?

Visualizing Space

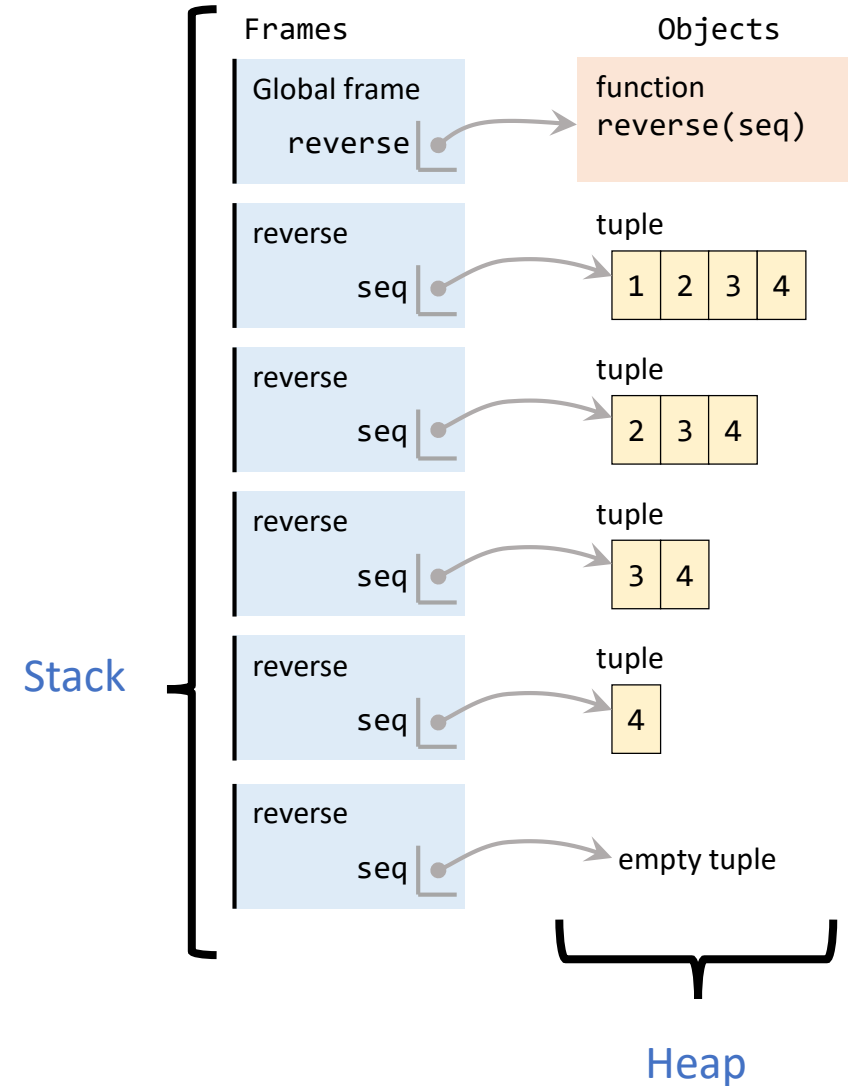
```
def reverse(seq):  
    if seq == ():  
        return ()  
    else:  
        return reverse(seq[1:]) +  
            (seq[0],)
```

`reverse((1, 2, 3, 4))`

Order of Growth

Time: $O(n + n^2) = O(n^2)$

Space: $O(n + n^2) = O(n^2)$



Orders of Growth

```
def reverse(seq):  
    result = ()  
    for item in seq:  
        result = (item,) + result  
    return result
```

Iterative

$\text{tuple1} + \text{tuple2}$ takes $\text{len}(\text{tuple1}) + \text{len}(\text{tuple2})$ steps!

- Orders of growth:

	Time	Space
- Recursive version:	$O(n^2)$	$O(n^2)$
- Iterative version:	$O(n^2)$	$O(n)$

Key Idea:

Handle the First Element
and then the Rest

Iterate/recurse down the sequence!

Scaling a sequence

Suppose we want to scale all the elements of a sequence by some factor

`scale_seq((1, 2, 3, 4), 3) → (3, 6, 9, 12)`

```
def scale_seq(seq, factor):
```

```
    if seq == ():
```

```
        return ()
```

```
    else:
```

```
        return (seq[0] * factor,) +
```

```
                scale_seq(seq[1:], factor)
```

Time? $O(n^2)$

Space? $O(n^2)$

Scaling a sequence (iterative)

Suppose we want to scale all the elements of a sequence by some factor

`scale_seq((1, 2, 3, 4), 3) → (3, 6, 9, 12)`

```
def scale_seq(seq, factor):  
    result = ()  
    for element in seq:  
        result = result + (element * factor,)   
    return result
```

Time? $O(n^2)$

Space? $O(n)$

Squaring a sequence

Given a sequence, we want to return a sequence of the squares of all elements.

`square_seq((1, 2, 3, 4))` \rightarrow `(1, 4, 9, 16)`

```
def square_seq(seq):
```

```
    if seq == ():
```

```
        return ()
```

```
    else:
```

```
        return (seq[0] ** 2, ) +  
                square_seq(seq[1:])
```

Time? $O(n^2)$

Space? $O(n^2)$

Homework: Do this iteratively

Looking for patterns

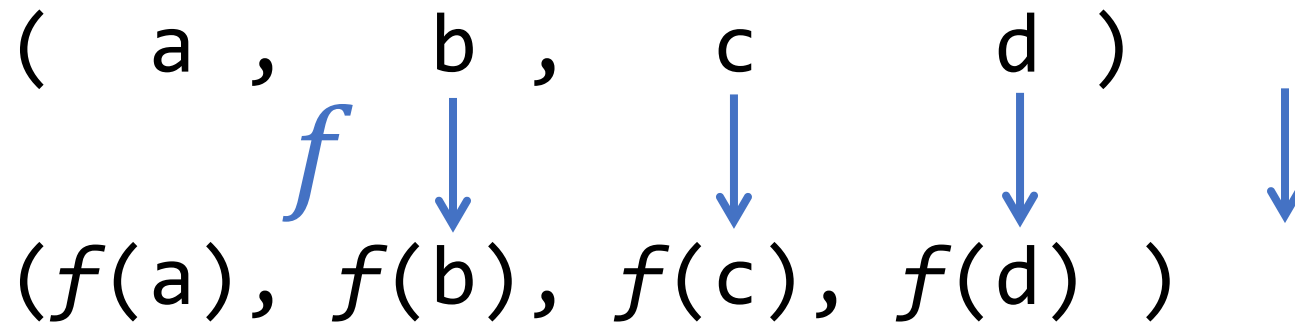
```
def scale_seq(seq, factor):  
    if seq == ():  
        return ()  
    else:  
        return (seq[0] * factor,) +  
                scale_seq(seq[1:], factor)
```

```
def square_seq(seq):  
    if seq == ():  
        return ()  
    else:  
        return (seq[0] ** 2,) +  
                square_seq(seq[1:])
```

Higher-order
function!!

Mapping

Often, we want to perform the same operation on every element of a list.



This is called *mapping*.

Mapping

```
def map(fn, seq):  
    if seq == ():  
        return ()  
    else:  
        return (fn(seq[0]), ) + map(fn, seq[1:])
```

Note: this will overwrite
the default Python map function!

Scaling a list by a factor

```
def scale_seq(seq, factor):  
    return map(lambda x: x * factor, seq)
```

Examples

`map(abs, (-10, 2.5, -11.6, 17))`

`→ (10, 2.5, 11.6, 17)`

`map(square, (1, 2, 3, 4))`

`→ (1, 4, 9, 16)`

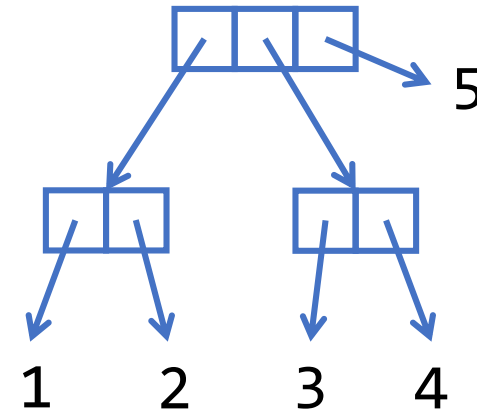
`map(cube, (1, 2, 3, 4))`

`→ (1, 8, 27, 64)`

Trees

Trees are sequences of sequences and single elements

- This is possible because of the closure property: we can include a sequence as an element of another sequence
- This allows us to build hierarchical structures, e.g. trees.
((1, 2), (3, 4), 5)



Examples

```
>>> x = ((1, 2), 3, 4)
```

```
>>> len(x)
```

```
3
```

```
>>> count_leaves(x)
```

```
4
```

```
>>> (x, x)
```

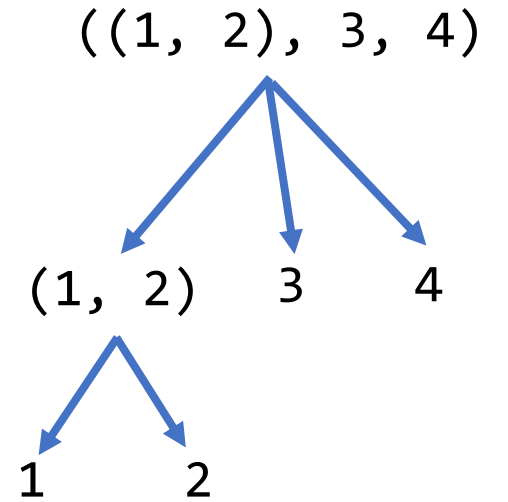
```
((1, 2), 3, 4), ((1, 2), 3, 4))
```

```
>>> len((x, x))
```

```
2
```

```
>>> count_leaves((x, x))
```

```
8
```

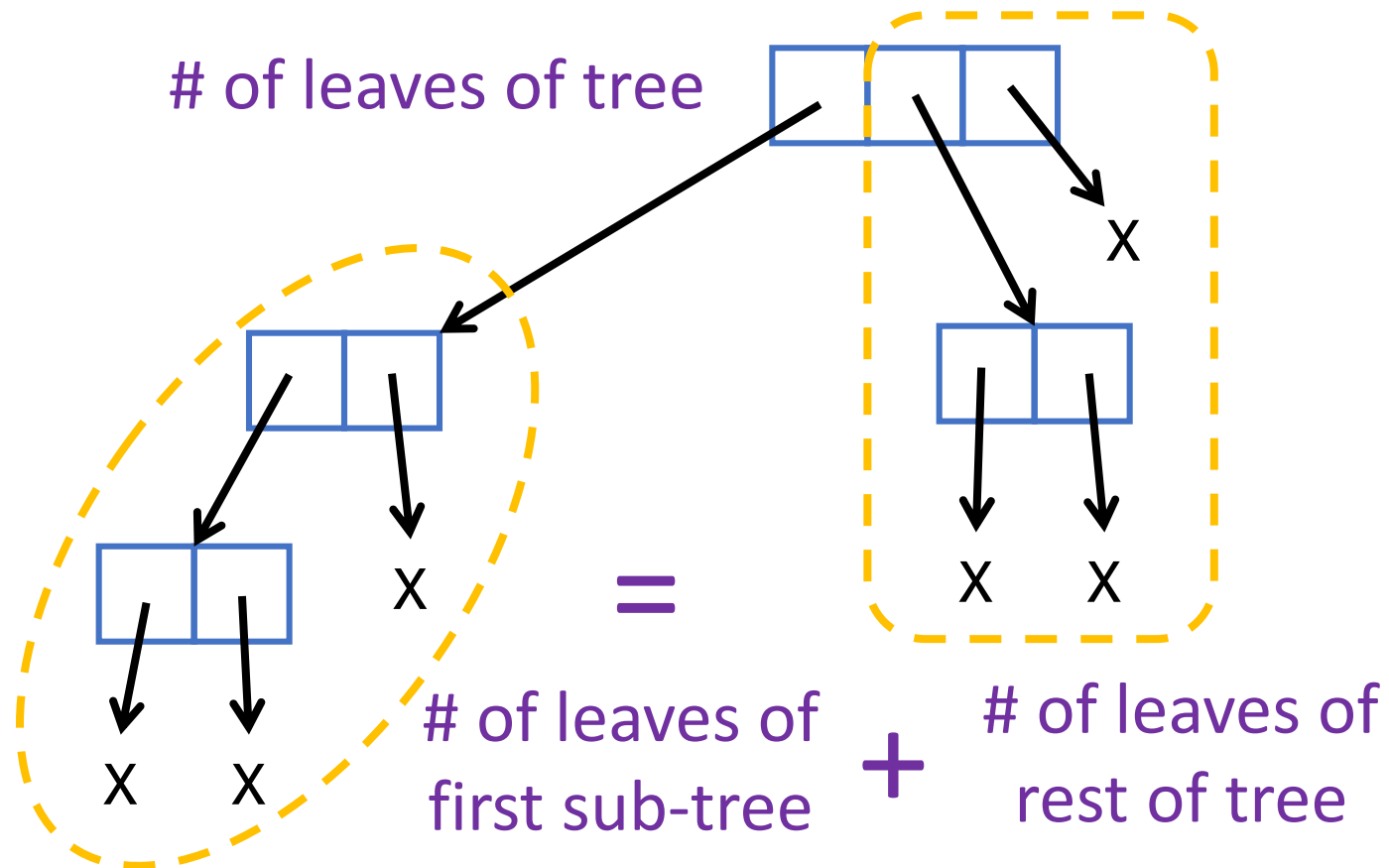


How would we count
the leaves?

RECURSION!

Recurrence Relation

Observation:



Recursion

In other words,

```
count_leaves(tree) =  
    count_leaves(tree[0]) +  
    count_leaves(tree[1:])
```

Base Case:

If tree is empty

Zero!

Another Base Case

Observe:

Possible for the head or tail to be a leaf!

Leaf $\Rightarrow +1$

Summary

Strategy:

- If tree is empty, then 0
- Another base case:
 - tree is a leaf, then count as 1
- Count this, and add to:
 - `tail` also a tree, so recursively count this

Count Leaves

```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
        + count_leaves(tree[1:])
```

What are leaves

Remember `type()` in Lecture 1:

```
>>> t = (1, 2, 3)
```

```
>>> type(t)
```

```
<class 'tuple'>
```

```
>>> type(t) == tuple
```

```
True
```

```
def is_leaf(item):
```

```
    return type(item) != tuple
```

Mapping over trees

Suppose we want to scale each leaf by a factor, i.e.

`mytree` \rightarrow `(1, (2, (3, 4), 5), (6, 7))`

`scale_tree(mytree, 10)`

\rightarrow `(10, (20, (30, 40), 50), (60, 70))`

Strategy

- Since tree is a **sequence of sequences**, we can map over each element in a tree.
- Each element is a subtree, which we recursively scale, and return sequence of results.
- **Base case:** if tree is a leaf, multiply by factor

Mapping over trees

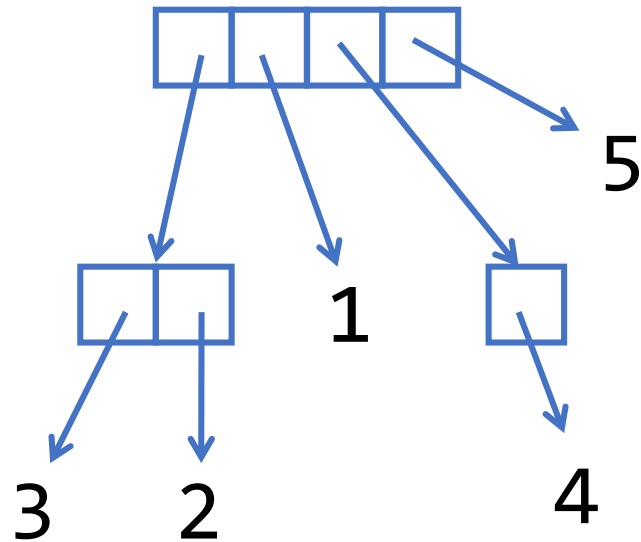
```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor * subtree  
        else:  
            return scale_tree(subtree, factor)  
    return map(scale_func, tree)
```

Compare with:

```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0]) + count_leaves(tree[1:])
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```



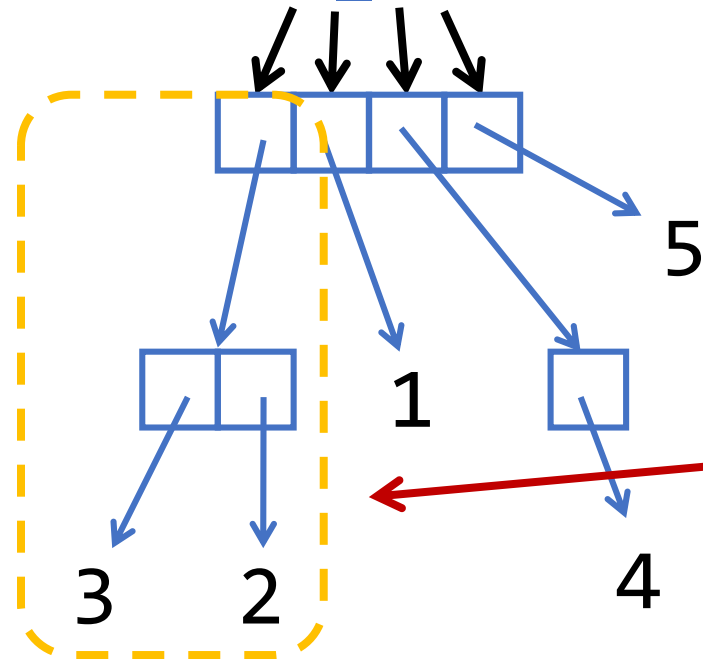
```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree, factor)  
    return map(scale_func, tree)
```

Suppose we do `scale_tree(tree, 2)`

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element



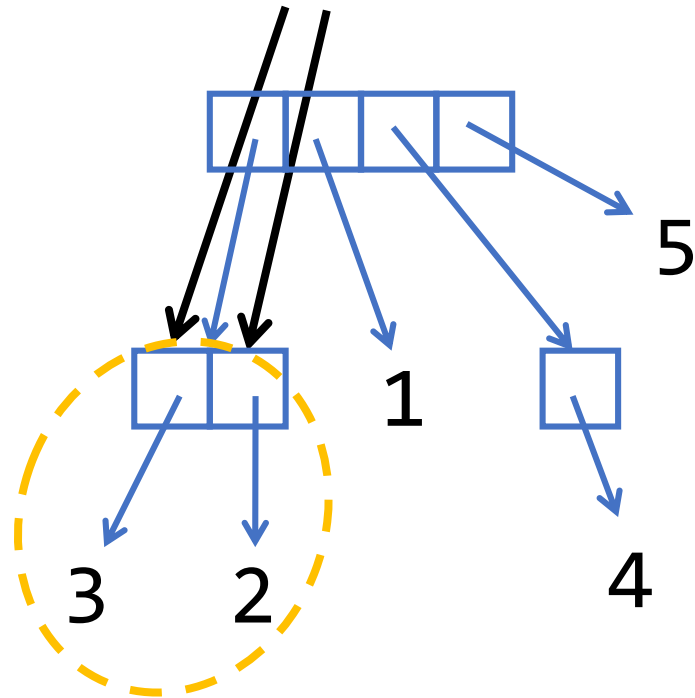
Not a leaf

```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree, factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

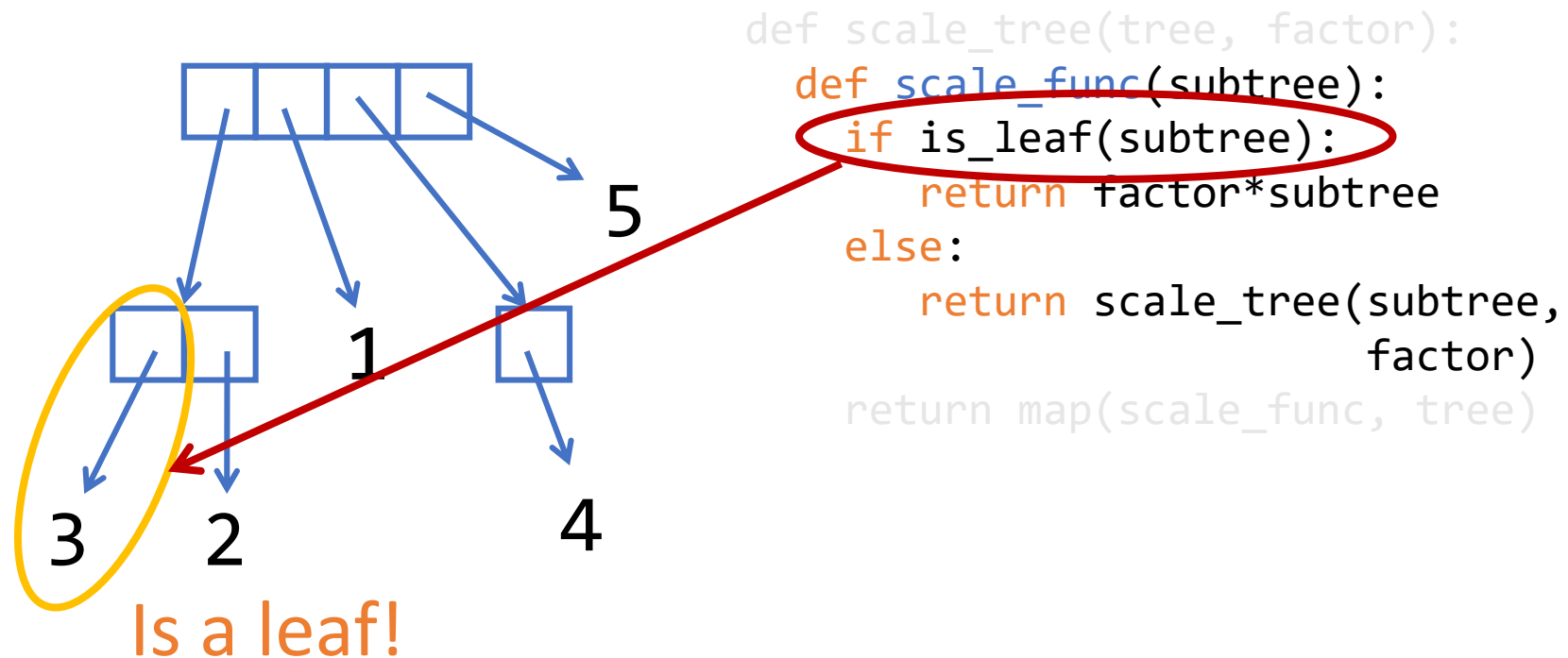
Apply `scale_func` to each element



```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

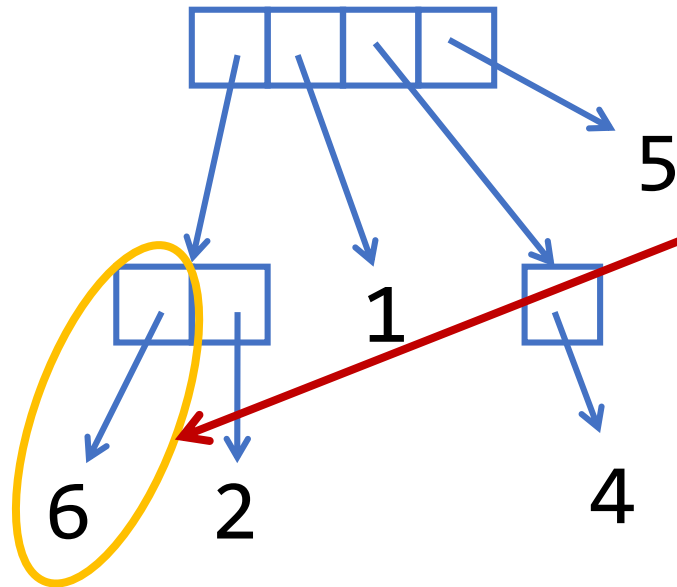

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```



Let's see what `scale_tree` does

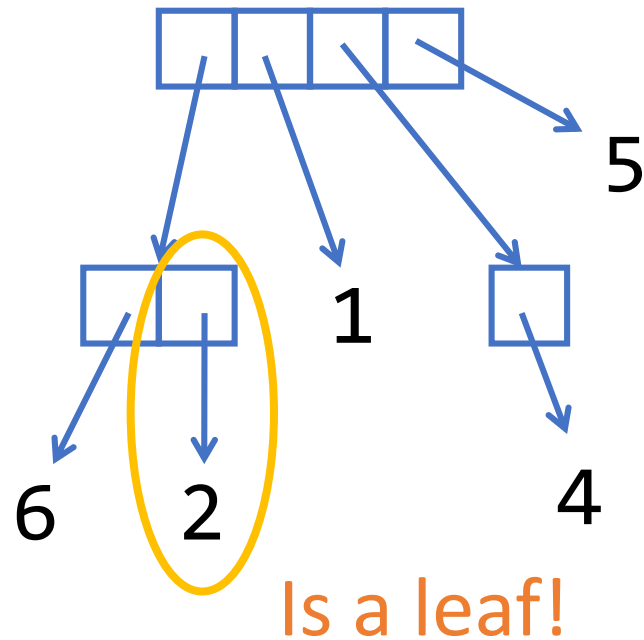
```
tree = ((3, 2), 1, (4,), 5)
```



```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

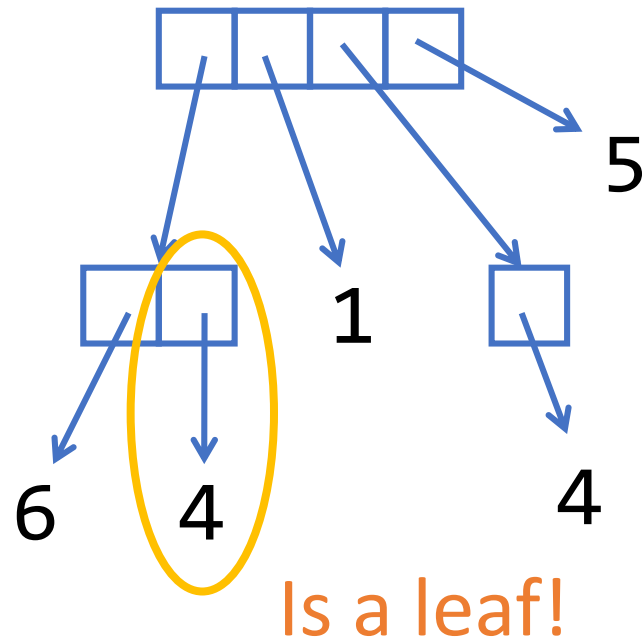
```
tree = ((3, 2), 1, (4,), 5)
```



```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

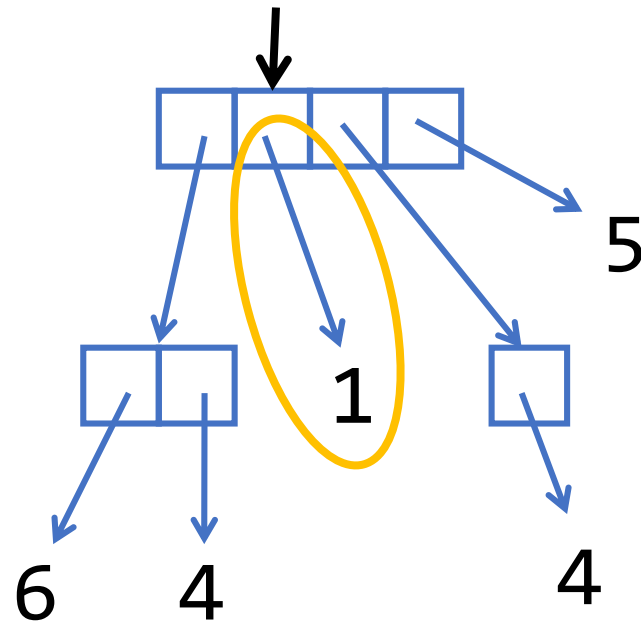


```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element



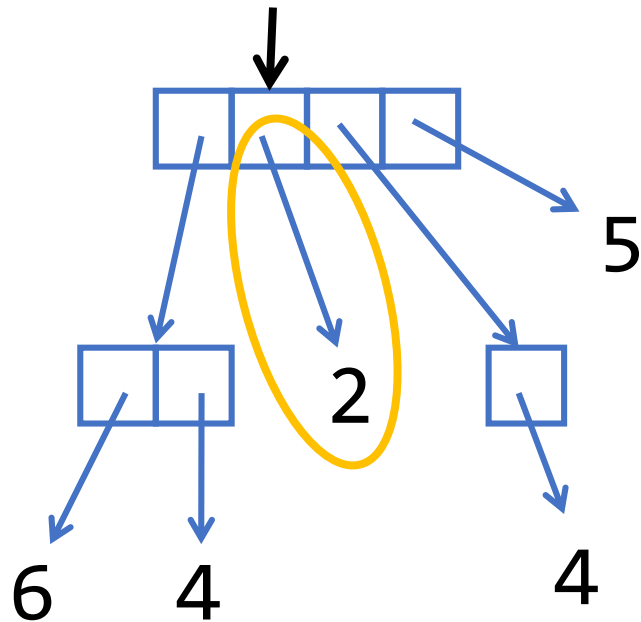
Is a leaf!

```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element

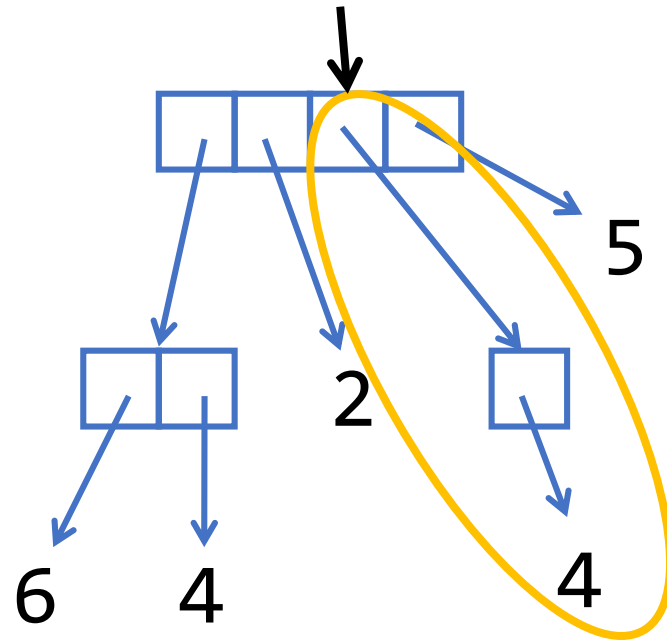


```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element



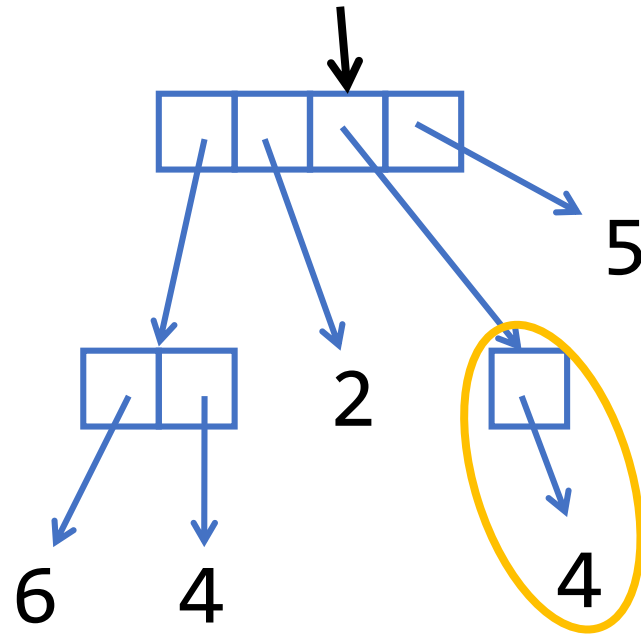
Not a leaf!

```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element

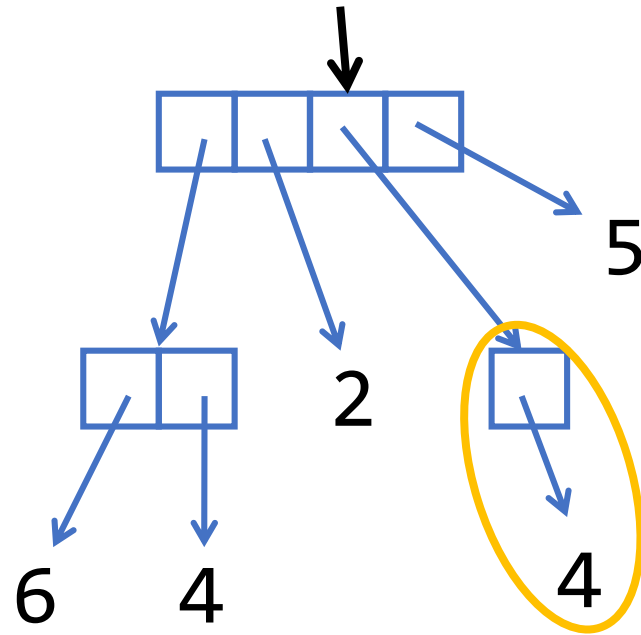


```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```


Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element



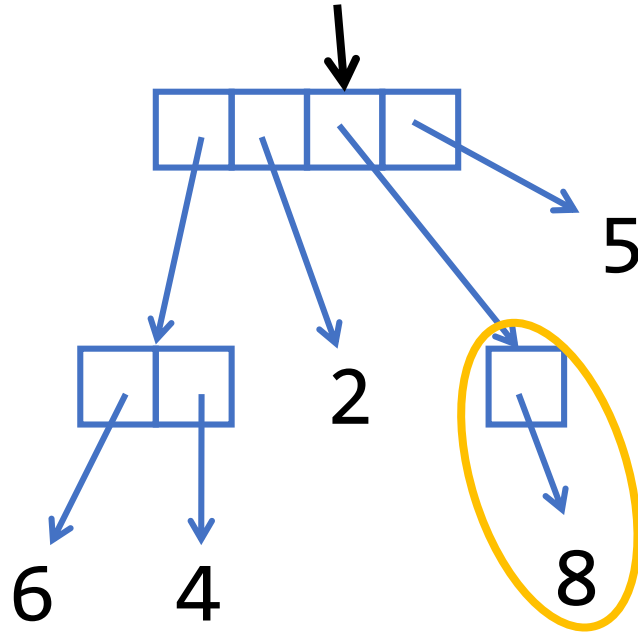
Is a leaf!

```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree, factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element

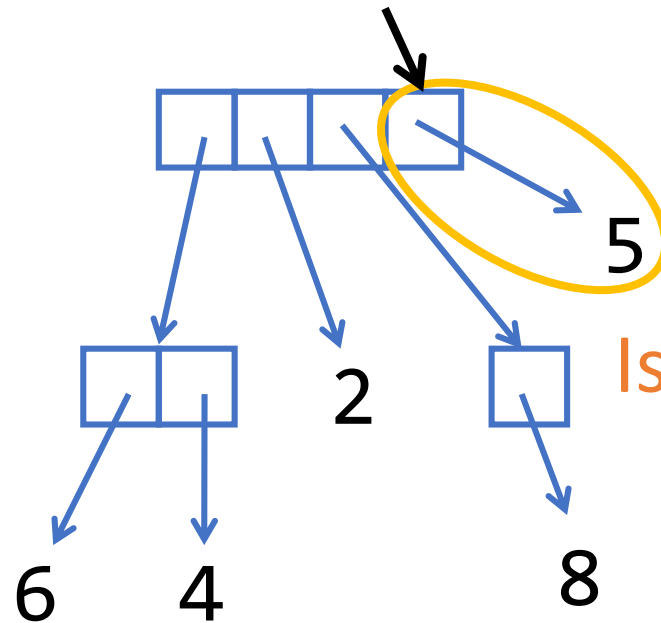


```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

Apply `scale_func` to each element



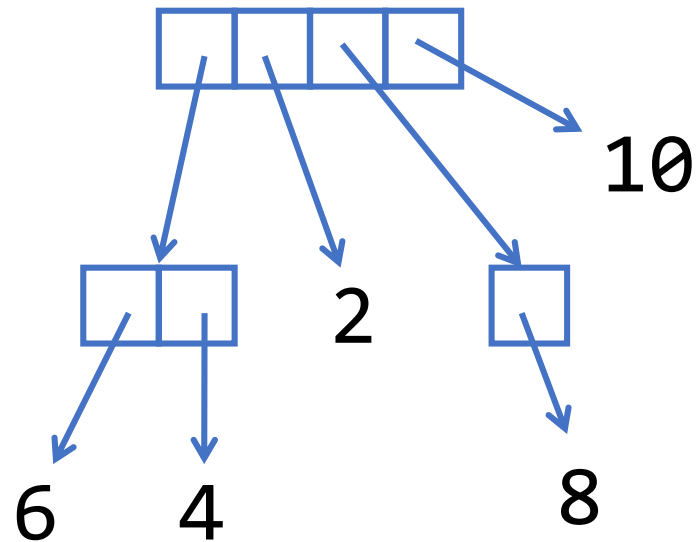
```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree, factor)  
    return map(scale_func, tree)
```

Is a leaf!

Let's see what `scale_tree` does

```
tree = ((3, 2), 1, (4,), 5)
```

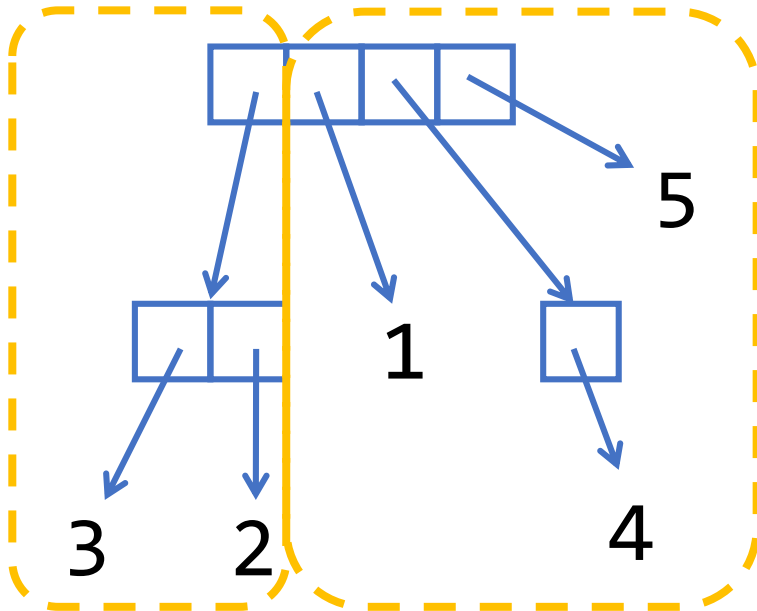
Done applying `scale_func` to each element



```
def scale_tree(tree, factor):  
    def scale_func(subtree):  
        if is_leaf(subtree):  
            return factor*subtree  
        else:  
            return scale_tree(subtree,  
                               factor)  
    return map(scale_func, tree)
```

Let's compare with `count_leaves`

`tree = ((3, 2), 1, (4,), 5)`

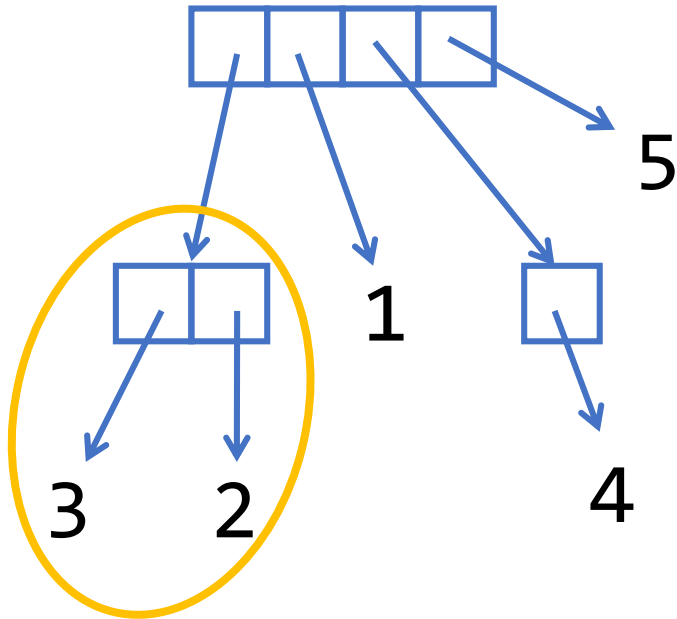


`count_leaves` + `count_leaves`

```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

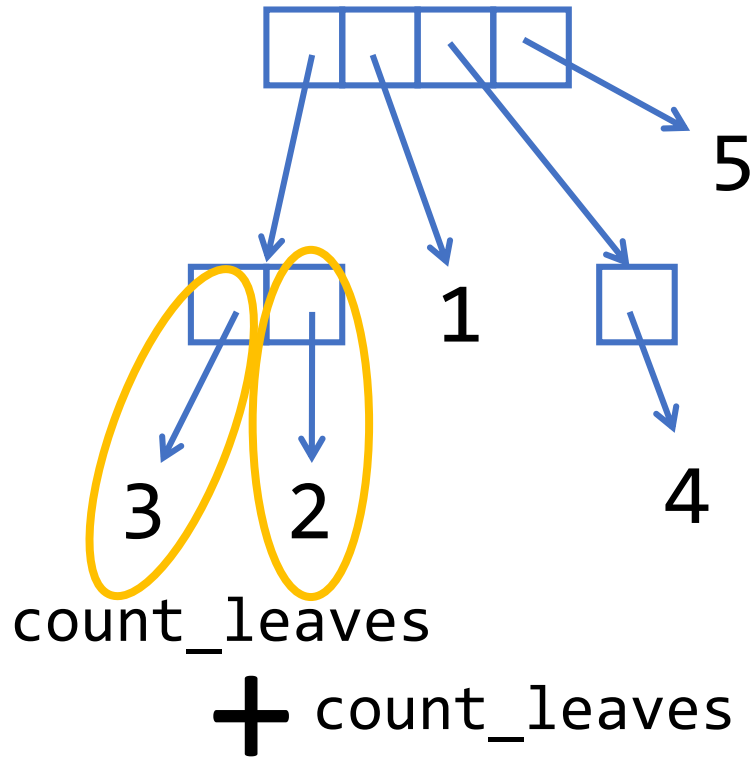
```
tree = ((3, 2), 1, (4,), 5)
```



```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

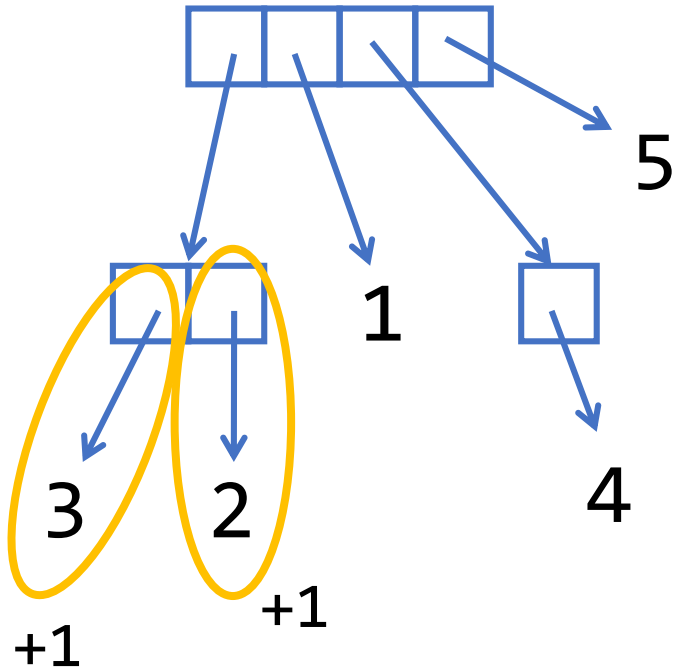
```
tree = ((3, 2), 1, (4,), 5)
```



```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

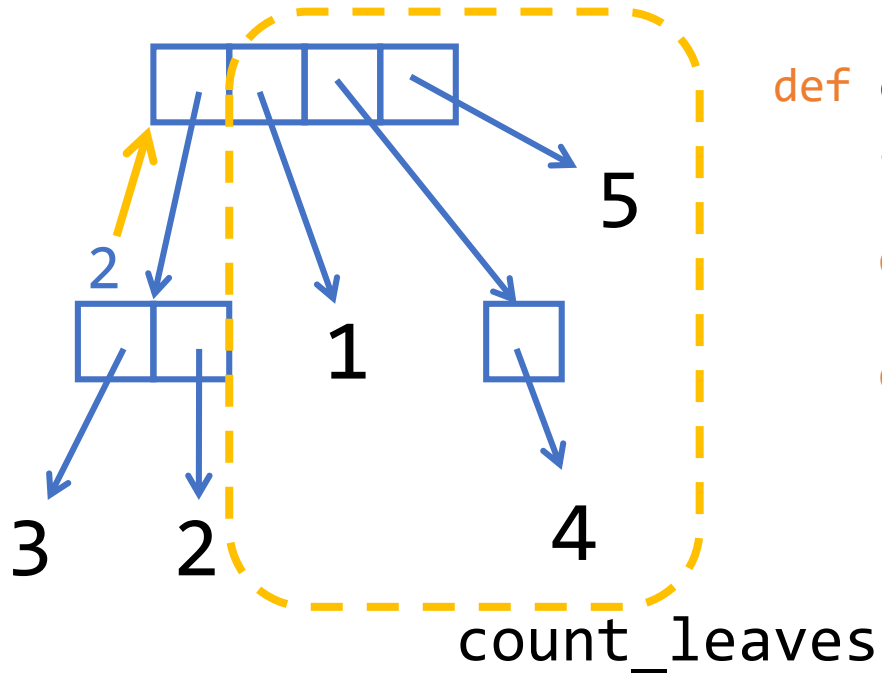
```
tree = ((3, 2), 1, (4,), 5)
```



```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```


Let's compare with `count_leaves`

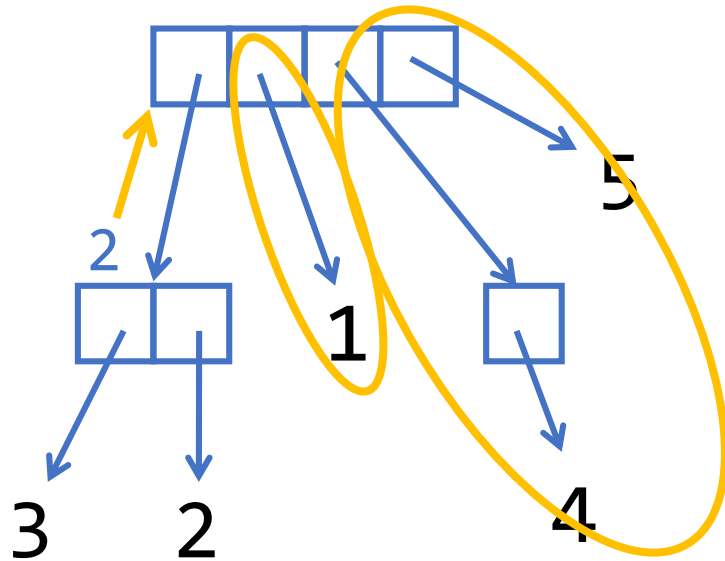
```
tree = ((3, 2), 1, (4,), 5)
```



```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

```
tree = ((3, 2), 1, (4,), 5)
```

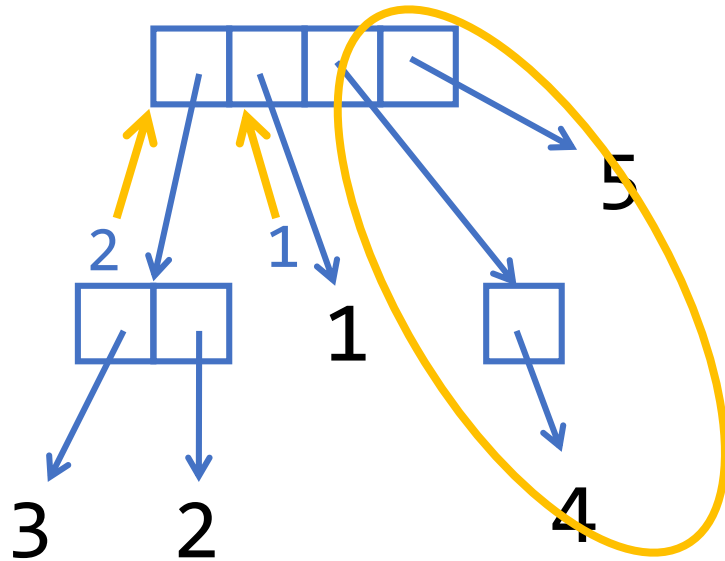


```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

`count_leaves` + `count_leaves`

Let's compare with `count_leaves`

```
tree = ((3, 2), 1, (4,), 5)
```

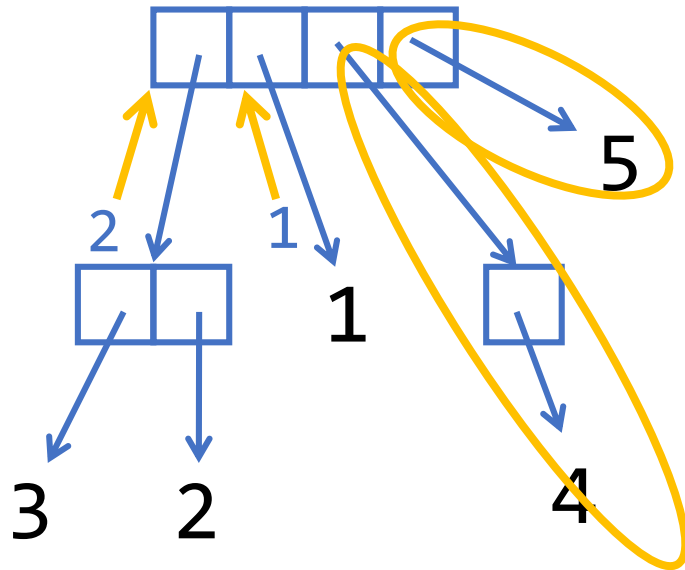


`count_leaves`

```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

```
tree = ((3, 2), 1, (4,), 5)
```

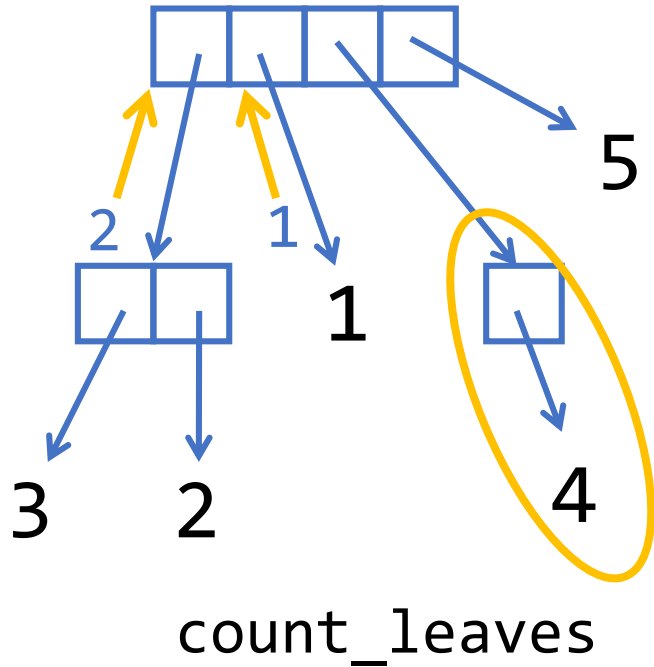


```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

`count_leaves` + `count_leaves`

Let's compare with `count_leaves`

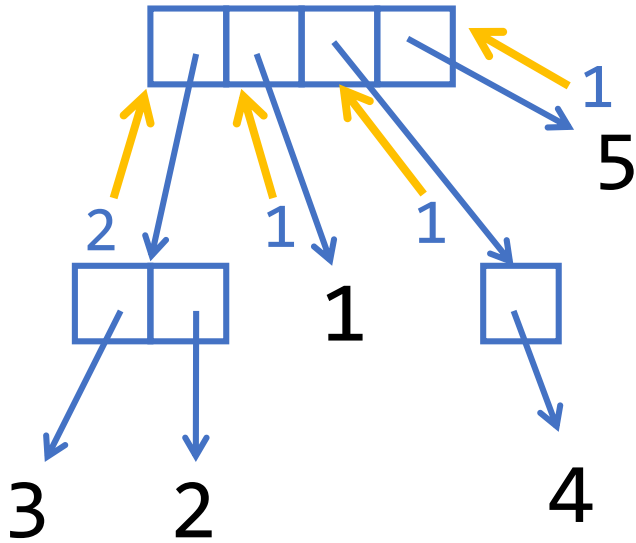
```
tree = ((3, 2), 1, (4,), 5)
```



```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Let's compare with `count_leaves`

```
tree = ((3, 2), 1, (4,), 5)
```



```
def count_leaves(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        return 1  
    else:  
        return count_leaves(tree[0])  
               + count_leaves(tree[1:])
```

Key Idea:

Traverse tree with recursion

Check for leaf!

Sanity Check (QOTD)

How do you write a function `copy_tree` that takes a tree and returns a copy of that tree?

Sanity Check (QOTD)

```
def copy_tree(tree):  
    return tree # is NOT acceptable!
```

```
>>> t = (1, 2, 3)
```

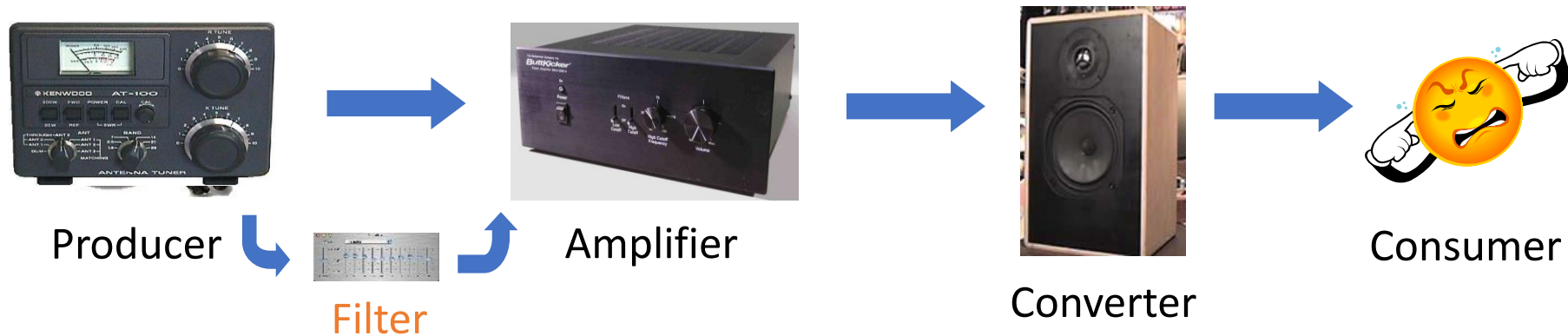
```
>>> t_copy = copy_tree(t)
```

```
t == t_copy → True
```

```
t is t_copy → False
```

Listening to Music

- Signal goes through various stages of “processing” .
- Additional component can be inserted.
- Easy to change component.
- Components interface via **signals**.



Modeling Computation as Signal Processing

- Producer (enumerator) creates signal.
- Filter removes some elements.
- Mapper modifies signal.
- Consumer (accumulator) consumes signal.

Benefits

1. Modularity: each component independent of others; components may be re-used.
2. Clarity: separates data from processes
3. Flexibility: new component can be added

Example:

Sum of squares of odd leaves

Given a tree, want to add the squares of (only) leaves of odd numbers:

`sum_odd_squares(((1, 2), (3, 4))) → 10`

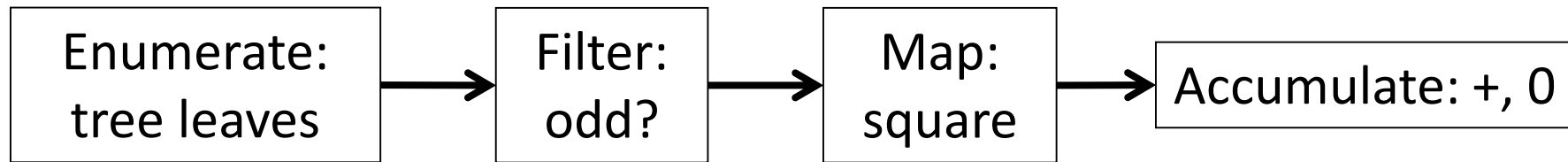
Example:

Sum of squares of odd leaves

```
def sum_odd_squares(tree):  
    if tree == ():  
        return 0  
    elif is_leaf(tree):  
        if tree % 2 == 0:  
            return 0  
        else:  
            return tree ** 2  
    else:  
        return sum_odd_squares(tree[0]) +  
               sum_odd_squares(tree[1:])
```

Alternative Approach

View it as signal processing computation!



How to represent “signals”?

- Sequences

Enumerating leaves

What does the following function do?

```
def enumerate_tree(tree):  
    if tree == ():  
        return ()  
    elif is_leaf(tree):  
        return (tree,)  
    else:  
        return enumerate_tree(tree[0]) +  
               enumerate_tree(tree[1:])
```

`enumerate_tree((1, (2, (3, 4)), 5))`

→ `(1, 2, 3, 4, 5)`

Also known as flattening the tree.

Filtering a sequence

```
def filter(pred, seq):
```

```
    if seq == ():
```

```
        return ()
```

Note: we are overwriting
the default Python filter function!

```
    elif pred(seq[0]):
```

```
        return (seq[0],)
```

```
            + filter(pred, seq[1:])
```

```
    else:
```

```
        return filter(pred, seq[1:])
```

```
is_odd = lambda x: x%2 != 0
```

```
filter(is_odd, (1, 2, 3, 4, 5)) → (1, 3, 5)
```

Accumulating a sequence

```
def accumulate(fn, initial, seq):  
    if seq == ():  
        return initial  
    else:  
        return fn(seq[0],  
                    accumulate(fn, initial,  
                               seq[1:]))
```

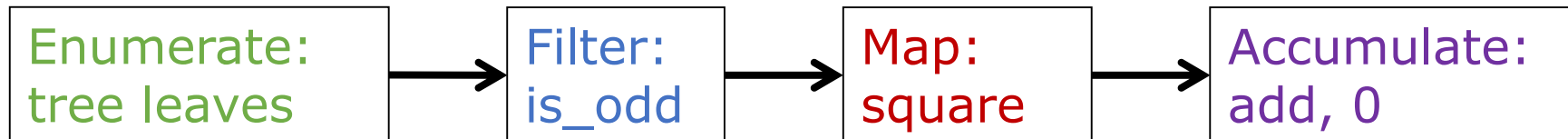
```
add = lambda x, y: x+y  
accumulate(add, 0, (1, 2, 3, 4, 5))  
accumulate(lambda x, y:(x, y), (),  
            (1, 2, 3, 4, 5))
```

```
→ (1, (2, (3, (4, (5, ())))))
```

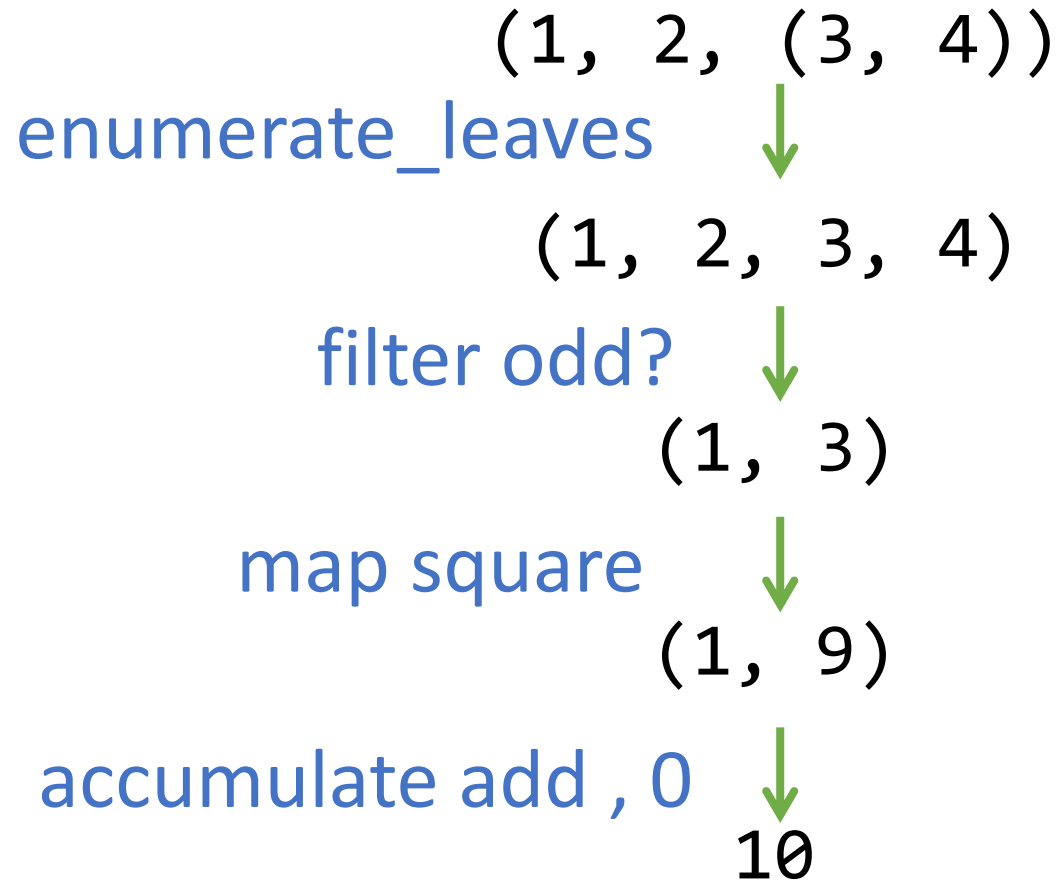
→ 15

Putting it together

```
def sum_odd_squares(tree):  
    return \  
    accumulate(add, 0,  
               map(square,  
                   filter(is_odd,  
                           enumerate_tree(tree))))
```



Putting it together



Another Example: Tuple of even Fib

Want a list of even *fib(k)* for
all *k* up to given integer *n*.

“Usual” Way

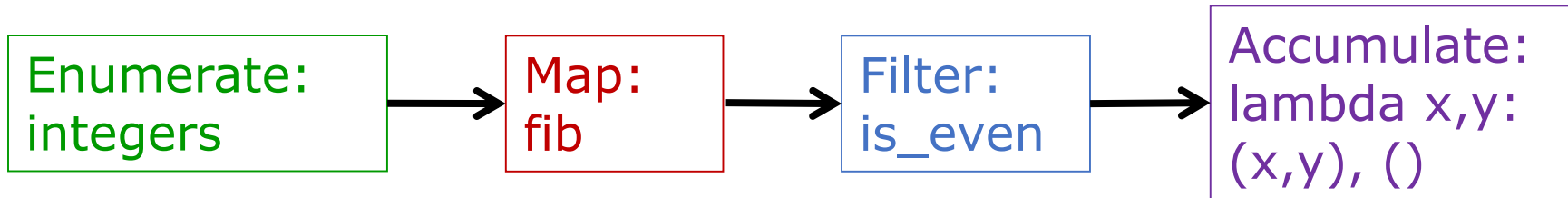
```
def even_fibs(n):  
    result = ()  
    for k in range(0, n + 1):  
        f = fib(k)  
        if is_even(f):  
            result = result + (f, )  
    return result
```

```
is_even = lambda x: x % 2 == 0
```

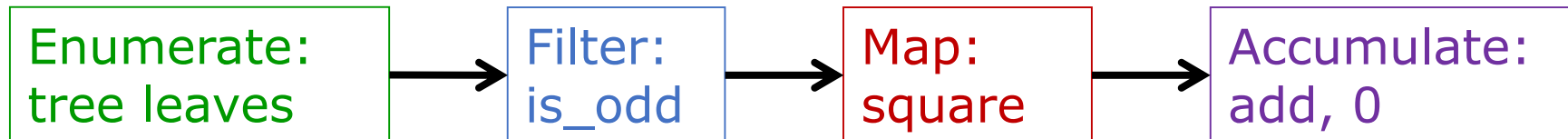
```
>>> even_fibs(30)  
(0, 2, 8, 34, 144, 610, 2584, 10946, 46368, 196418, 832040)
```

Signal processing view

- Even fib



- Compare: sum square odd leaves



Enumerate integers

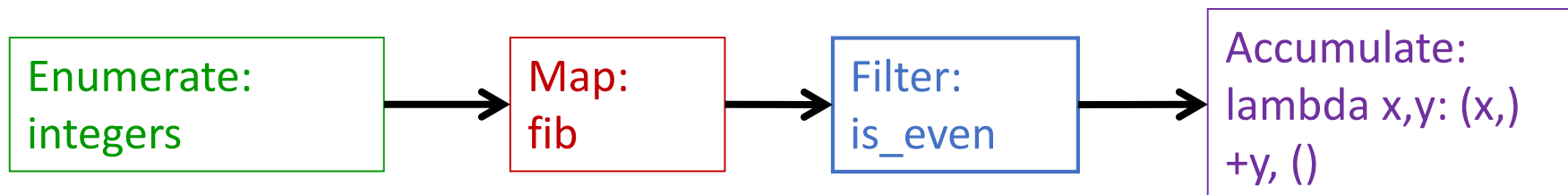
```
def enumerate_interval(low, high):  
    return tuple(range(low, high+1))
```

```
enumerate_interval(2, 7)
```

```
→ (2, 3, 4, 5, 6, 7)
```


Even Fibs

```
def even_fibs(n):  
    return  
        accumulate(lambda x, y: (x,) + y, (),  
                    filter(is_even,  
                            map(fib,  
                                enumerate_interval(0, n))))
```



Signal Processing View

- Modular components:
 - Enumerate, Filter, Map, Accumulate
 - Each is independent of others.
 - Modularity is a powerful strategy for controlling complexity.

Signal Processing View

- Build a library of components.
- Sequences used to interface between components.

Other Uses

- Suppose we have a sequence of personnel records.
- Want to find salary of highest-paid programmer.

```
def salary_of_highest_paid_programmer(records):  
    return accumulate(max, 0,  
                       map(salary,  
                           filter(is_programmer,  
                                records)))
```

Default Python `map` and `filter` functions

Returns an “iterable” instead of tuple, but you can force it into a tuple.

```
>>> a = (1, 2, 3, 4, 5)
>>> b = filter(lambda x: x%2 == 0, a)
>>> b
<filter object at 0x02EC4710>
```

Think of iterable as a “one-time” use
sequence

```
>>> b  
<filter object at 0x02EC4710>
```

```
>>> for i in b:  
    print(i)
```

```
2
```

```
4
```

```
>>> tuple(b)  
( )
```



```
>>> b = filter(lambda x: x%2 == 0, a)
```

```
>>> b
```

```
<filter object at 0x02E42C10>
```

```
>>> c = tuple(b)
```

```
>>> c
```

```
(2, 4)
```

```
>>> tuple(b)
```

```
()
```

Comprehension

If `map` and `filter` is too complicated...

`(<expr> for <var> in <seq> if <cond>)`

is equivalent to

```
map(lambda <var>: <expr>,  
    filter(lambda <var>:<cond>, <seq>))
```

```
a = (1,2,3,4,5)
```

```
>>> b = tuple((x*2 for x in a))
```

```
>>> b
```

```
(2, 4, 6, 8, 10)
```

$2x, \forall x \in a$

```
>>> b = tuple((x*2 for x in a if x%2 == 0))
```

```
>>> b
```

```
(4, 8)
```

$2x, \forall x \in a, x \bmod 2 = 0$

Working with Files

Reading a File:

```
input = open('inputfilename.txt', 'r')  
some_line = input.readline()
```

We can check for end of file by checking whether

```
some_line == '' #empty string
```

Writing to a File:

```
output = open('outputfilename.txt', 'w')  
output.write('HELLO WORLD')
```

Example

```
def metrics(dictfile):  
    dict = open(dictfile, 'r')  
  
    currword = dict.readline()  
    longest_word = currword  
    shortest_word = currword  
  
    while currword != '':  
        if(len(currword) < len(shortest_word)):  
            shortest_word = currword  
        if(len(currword) > len(longest_word)):  
            longest_word = currword  
        currword = dict.readline()  
  
    output = open("output.txt", "w")  
    output.write("longest word: " + longest_word)  
    output.write("shortest word: " + shortest_word)
```



Find longest and
shortest word



write to file

Example

```
dictionary.txt >>
```

```
CS1010S
```

```
BEST
```

```
MODULE
```

```
WORLD
```

```
metrics("dictionary.txt")
```

```
output.txt >>
```

```
longest word: CS1010S
```

```
shortest word: BEST
```

Summary

- Data often comes in the form of sequences
 - Easy to manipulate using recursion/iteration
 - Can be nested
- Closure property allows us to build hierarchical structures, e.g. trees, with tuples
 - Can use recursion to traverse such structures

Summary

- “Signal-processing” view of computation.
 - Powerful way to organize computation.
 - Sequences as interfaces
 - Components: (i) Enumeration, (ii) Map, (iii) Filter, (iv) Accumulate